AUEB M.Sc. in Data Science (part-time)

Course: Text Analytics

Semester: Spring 2019

Title: Report of Homework 01

Team members:

- Alexandros Kaplanis (p3351802)
- Spiros Politis (p3351814)
- Manos Proimakis (p3351815)

Introduction and motivation

The purpose of this homework is to explore n-gram language modelling techniques covered in the first lecture of the Text Analytics course. The language models used are bigrams and trigrams, for which we use their respective quality metrics, namely cross-entropy and perplexity to measure their effectiveness.

Execution instructions

Python environment setup

Create a Conda virtual env:

```
conda create -n msc-ds-ta-homework-1 python=3.6
```

Activate the virtual env:

```
source activate msc-ds-ta-homework-1 (Linux)
activate msc-ds-ta-homework-1 (Windows)
```

Install required Python packages:

```
pip install -r requirements.txt
```

Execution

1. Providing your own sentences in / out of the corpus

Use this option to provide the known / unknown sentences that will be used for the evaluation of the language models.

```
python homework-1-exersice-4-main.py -v -file europarl-v7.el-en.en -num_lines 10
00 -train_test_split_percentage 0.8 -sentence_in_corpus "This sentence is in cor
pus" -sentence_not_in_corpus "This sentence is not in corpus"
```

2. Picking / generating random sentences from the corpus

Use this option to let the program pick at random a sentence from the corpus and use that same sentence to also produce a sentence not present in the corpus.

```
python homework-1-exersice-4-main.py -v -file europarl-v7.el-en.en -num_lines 10
00 -train_test_split_percentage 0.8
```

Methodology

Dataset

For a corpus, we used the English part of Europarl (available from http://www.statmt.org/europarl/ (http://www.statmt.org/europarl/)), specifically the file *europarl-v7.el-en.en* (available from http://www.statmt.org/europarl/v7/el-en.tgz (http://www.statmt.org/europarl/v7/el-en.tgz)). Since the corpus is roughly 192 MB and comprises about 1.25 M lines, we implemented a function allowing us to select a subset of the entire corpus, mainly for computational / performance purposes. To this end, we opted for a corpus of 100K sentences making sure, though, that the corpus splitting process is easily parameterized, meaning that we can increase / decrease our set at will.

Implementation details

Note: Although initially we chose to hand-code the required mechanics (n-gram building, language model classes etc.) for learning purposes and reached a sufficiently good and performant implementation, we decided to proceed with implementing our work using standard Python libraries and packages, most notably *NLTK*. The reasoning behind our choice is that we opted for an industry-adopted framework, strengthening our knowledge not only on the theoretical aspects of the task at hand but also to standardized practices at large.

Word sequencing

We experimented with the use of two different word sequences, one being sentences and the other line breaks. Given the better overall score however, we decided to go with sentences.

Dataset split

We used 80% of the dataset's **sentences** for the training process and the rest 20% for testing. The reasoning for partitioning the dataset using sentences instead of words was to avoid ending up with incomplete sentences.

We implemented all related dataset manipulation functionality in the function

```
def split(corpus, percent = 0.8, shuffle = False)
```

of *homework-exersice-1-main.py*.

Padding

Since the original padding that is provided by *NLTK* did not match the following requirements:

- Adding indexes on the padding e.g *start1*, *start2*, ... *startN*
- Adding single *end* padding. (*NLTK* includes the same number of end padding as the start)

we decided to create a helper class (*SentencePadding*) that would allow us to have this functionality.

Vocabulary

As per the exercise's instructions we generated the vocabulary based on words that appear at least 10 times in the corpus. All other words are categorized with the token "UNK". It is worth mentioning that we considered all punctuation marks a part of the vocabulary represented by the same "word".

All related functionality is implemented in class *Vocabulary*.

Smoothing

We used Laplace smoothing which estimates the propability of a *n-gram* based on

$$P(ngram) = \frac{Count(ngram)+a}{Count((n-1)gram)+a*Count(Vocabulary)}$$

- On the above equation the vocabulary includes only the unique words that had a count greater that 10 in the train corpus.
- The **Laplace smoothing** is a form of **Lidstone smoothing** were the **a** hyperparameter is **1**.

Language Model

We have created a separate Language Model class to host the functionality required to train a model and estimate the propability of a sentence to have come from the corpus.

Initialization

First of all the Language Model is initialized with:

- a **Vocabulary** trained from the corpus hosting all the unique words of the corpus that had a count above a threshold which on our case is **10**.
- a **Padding helper** that will manage the padding addition
- the **a** smoothing parameter set to 1 for laplace

Training Procedure

In the training step we will execute the functionality required to train the model and allow for future probability prediction:

1. The corpus is **normalized**, all text is coverted to lower (same step was done to vocabulary training too) so as to treat words regardless of their case.
2. The corpus is split into complete sentences.
3. Each sentence is cleaned (all words that do not exist in the vocabulary are beign replaced with ***UNK***).
4. A correct Padding is added to the sentence according to the ngram rank e.g *start1*, *start2* for the trigram and *start1* fro the bigram
5. The sentence is tokenized using the TweetTokenizer which worked best after our tests. (We have also tried WordTokenized and RegexTokenizer provided by *NLTK*)
6. We split to ngrams of the models rank.
7. And we update two Counters that we used to host the Frequencies of each ngram in the sentence (including the pad symbols).

Prediction

In the prediciton process we estimate the $log_2(prob)$ of a sentence based on the fitted model:

(1), (2) and (3) steps are the same as in the training so the sentence is tranformed to the model baseline.

4) After that we split the sentence into n-grams of the Language Model rank.

5) For each tuple we calculate the proability based on the Frequencies we calculated on the training process and using **Laplace Smoothing** defined above

6) All tuple propabilities are summed and the result is the estimated propability of the sentence, based on the training data.

$$Prediction(sentence) = \sum_{i=1}^{N} \log_2 p\left(ngram_i\right)$$

Scoring

To calculate the efficiency of the trained language model, we calculated the **entropy** and **perplexity** on a test set.

Similar to the prediction process we have the same (1), (2), (3) (4) steps.

But with the important change, that we calculate the sum for the whole corpus and devide by the number of ngrams

- to calculate the cross entropy of the model on the test set based on the below equation

  $$crossEntropy(testCorpus) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 p\left(ngram_i\right)$$

  where N is the number of ngrams after adding padding on the sentence.

- And the perplexity based on the below equation

  $$perplexity = 2^{crossEntropy}$$

  where cross-entropy is on the whole test corpus

Linear interpolation

Our goal for this part is to identify a good enough mix (i.e. going further from just plugging-in a set of random numbers) of parameters λ_1 and λ_2, such that our linearly interpolated model would produce the best possible results, without being exhaustive. To this end, we implemented the function

```
def compute_interpolation_vector(size = 10)
```

which allows us to retrieve N-tuples, $N \in \mathbb{R}_+$, of λ parameters of the model, satisfying the constraint $\lambda_1 + \lambda_2 = 1$. The constraint is implemented by selecting λ_1 from a uniform distribution and setting $\lambda_2 = 1 - \lambda_1$.

We chose to perform a modest $N = 10$ iterations and save the cross-entropy score of the linearly interpolated model to a Numpy array, allowing us to retrieve the index with the lowest cross-entropy score, which also coincides with the index of the $\boldsymbol{\lambda}$ parameter vector.

Observations and remarks

Metrics

Cross entropy has the following properties:

- The cross entropy $H(p, m)$ is an upper bound on the true entropy H(p).
- The closer the cross entropy H(p, m) is to the true entropy H(p), the more accurate the model m is.
- Cross entropy can therefore be used to compare approximate models Between two models m1 and m 2, the more accurate model will be the one with the lower cross entropy.

Perplexity is a good metric to measure how well a trained propability model or Language Model performs on predicting a sample. In practise we can use It to compare the Language Models. A low perplexity indicates the probability distribution is good at predicting the sample.

Language model performance

Bigram Model Performance:

- Cross Entropy: 6.913
- Perplexity: 120.480

Trigram Model Performance:

- Cross Entropy: 8.093
- Perplexity: 273.025

Combined Model Performance (Best cross-entropy score, parameters $\lambda_1 = 0.002, \lambda_2 = 0.997$):

- Cross Entropy: 0.0511
- Perplexity: 1.036
